hashbrown

Developed by Hyungho Chris Choi

Documented by Hyungho Chris Choi

Introduction

This program was proposed and developed in order to help the solution and representation of the multivariable first-order differential equations. The program embeds a Runge-Kutta Fourth order solution Method

The author of this program had inspiration from the Mathematical Modeling and Differential Equations class from the Korea Science Academy. And this program is conversely fit for students looking for a reliable solution software.

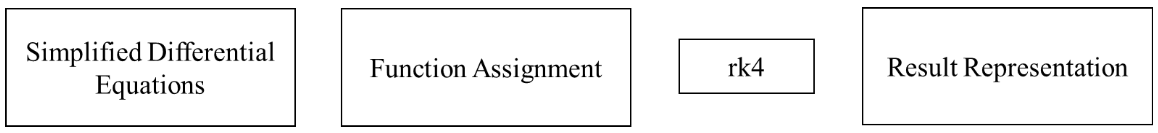
Special Thanks to prof. Y.D.Kim for the intellectual aid.

Documentation

Please read and follow the instructions on <https://pypi.org/project/hashbrown/> for proper installation and use.

Overview

The overall system workflow



This documentation will be based around an example of a forced oscillator, to easily describe the workflow.

0. Simplification

Before you look for the solution, the differential equation must be simplified to a simple multivariable first-order differential equation which has the form

$$\mathrm{x}_1' = f_1(t, x_1, x_2, \dots, x_n)$$

$$\mathrm{x}_2' = f_2(t, x_1, x_2, \dots, x_n)$$

$$\dots$$

$$\mathrm{x}_\mathrm{n}' = f_n(t, x_1, x_2, \dots, x_n)$$

The number of equations is irrelevant as long as the equations are given as the form above

For example, consider the forced harmonic equation

$$\mathrm{mx}'' + \mathrm{bx}' + \mathrm{kx} = F_0 \sin(wt)$$

We could simplify this by appointing $\mathrm{y} = \mathrm{x}'$ and concluding with a series of equations given as

$$\mathrm{x}' = y$$

$$\mathrm{y}' = -\frac{b}{m}y - \frac{k}{m}x + \frac{\mathrm{F}_0}{m}\sin(wt)$$

This process must be done by the user, the program can NOT (and should not) handle the simplification. You should also have the initial values ($\mathrm{x}_1(0), x_2(0) \dots \mathrm{x}_\mathrm{n}(0)$) Figured out accordingly.

1. Assigning Function Class Variables

Now the coding begins. Before we start, all you need to import is the **`hashbrown`** library. The library imports its dependencies in a hierarchy.

```
from hashbrown import *
```

From the simplified solutions from section 0, we assign **`[Function]`** class variables for each equation.

```
class Function:
   def __init__(self, f, var, name):
```

`f` is a python function that governs each equation. **`var`** is the number of variables that the function takes (including t) **`name`** is the name of this variable. The name parameter is very important, since it will be used in all result handling. ***Keep in mind, the name*** `'t'` ***is reserved for*** `hashbrown` ***internal (variable for time).***

`f` could be separately defined and then plugged into the **`[Function]`** class constructor.

Following with the example from section 0, the **`[Function]`** class variables could be defined as follows

```
from hashbrown import *
b = 1
m = 1
k = 1
F0 = 1
w = 10
def xp(t,x,y):
   return y
def yp(t,x,y):
   return -(b/m)*y - (k/m)*x + (F0/m)*sin(w*t)
F1 = Function(xp,3,'Displacement')
F2 = Function(yp,3,'Speed')
```

Make good use of the global variables for now. They will be encapsulated later.

Make sure you match the number of variables with all the functions. Even with functions like `xp`, we need all three parameters mentioned

We could also get creative (and nonlinear) with the functions: Suppose the force is no longer applied after `tf`. Then we could define

```
from hashbrown import *
b = 1
m = 1
k = 1
F0 = 1
w = 10
tf = 10
def F(t):
   if t>=tf:
      return 0
   return F0
def xp(t,x,y):
   return y
def yp(t,x,y):
   return -(b/m)*y - (k/m)*x - (F(t)/m)*sin(w*t)
```

2. Obtaining result dataset

The Runge-Kutta function (**`rk4`**) generates the solution dataset for the equation. It takes in four major variables

`rk4(ts, te, step, s, name)`

`ts, te, step (number)` : denotes initial, final time and timestep for runge-kutta method.

`name (string)`: the desired name of the result. Used in constructing filenames and plot titles.

`s (tuple)`: The most crucial parameter that defines the solution metrics. Explained further below.

`s` takes a tuple of tuples ***`( (values) , (functions) ).`*** Each element in ***`values`*** are the initial values of each variable, and each element in ***`functions`*** takes a **`[Function]`** class element. *It is very important that the initial values are in the same order as functions.*

Continued from the above example (the force stopping after `tf`) Let's generate a numerical solution. Where the initial state is at equilibrium (x,y are both zero)

```
ts = 0
te = 20
R = rk4(ts,te,0.0001,((0,0),(F1,F2)),'Forced Oscillation')
```

Now, R contains the result for the differential equation. Which as another custom class object from `hashbrown`.

3. Result Representation

The result obtained from (2.) is a **`[Result]`** class object. This class contains multiple methods to represent data.

3.0. Simple Methods (print data on shell)

`print`

simply printing the object prints the whole dataset in line order (in some IDE, the data will get omitted)

`.get(t,varN='')`

`.get` method returns the closest dataset to the desired time entered. Putting in just **`t`** will get all the parameters at that time(tuple), and specifying **`varN`** will get the result in that specified time in that specified variable

Continuing from section 2,

```
print (R.get(10))        #(9.999999999990033, [-0.004728505645778045, 0.09209838842791386])
print (R.get(10,'Speed'))# 0.09209838842791386
```

3.1. Export Data

There are a number of ways to export data out of the program

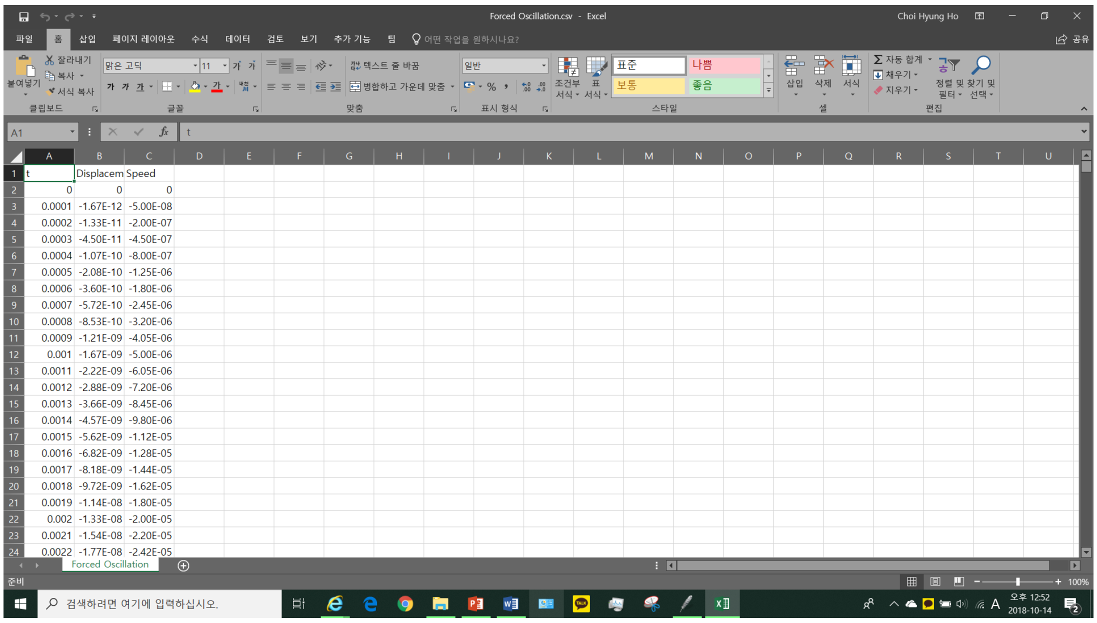
`.writeCSV(s=1)`

This method exports the dataset into the current working directory. Parameter `s` denotes how much data to actually save. **`.writeCSV()`** will save everything, and **`.writeCSV(10)`** will save every 10th data point. **`writeCSV(10)`** is recommended since the saving of CSV and EXCEL files are a long process

`.writeXLSX(s=1)`

Same deal with excel. Note that saving the excel file takes much longer than that of CSV. So use this when the datasets are small. Saving to CSV and converting to EXCEL is recommended (Until you have extra knowledge about **`openpyxl`**).

Continuing from section 2, **`R.writeCSV()`** saves as follows.



	A	B	C
1	t	Displacem	Speed
2	0	0	0
3	0.0001	-1.67E-12	-5.00E-08
4	0.0002	-1.33E-11	-2.00E-07
5	0.0003	-4.50E-11	-4.50E-07
6	0.0004	-1.07E-10	-8.00E-07
7	0.0005	-2.08E-10	-1.25E-06
8	0.0006	-3.60E-10	-1.80E-06
9	0.0007	-5.72E-10	-2.45E-06
10	0.0008	-8.53E-10	-3.20E-06
11	0.0009	-1.21E-09	-4.05E-06
12	0.001	-1.67E-09	-5.00E-06
13	0.0011	-2.22E-09	-6.05E-06
14	0.0012	-2.88E-09	-7.20E-06
15	0.0013	-3.66E-09	-8.45E-06
16	0.0014	-4.57E-09	-9.80E-06
17	0.0015	-5.62E-09	-1.12E-05
18	0.0016	-6.82E-09	-1.28E-05
19	0.0017	-8.18E-09	-1.44E-05
20	0.0018	-9.72E-09	-1.62E-05
21	0.0019	-1.14E-08	-1.80E-05
22	0.002	-1.33E-08	-2.00E-05
23	0.0021	-1.54E-08	-2.20E-05
24	0.0022	-1.77E-08	-2.42E-05

Notice that the names of each variable (defined in **`Function`** class) are already in place.

3.3. Graphic representation

There are ways to immediately represent the data graphically. This uses matplotlib.

`.writePLOT(subj = None, points = 1000,s=1)`

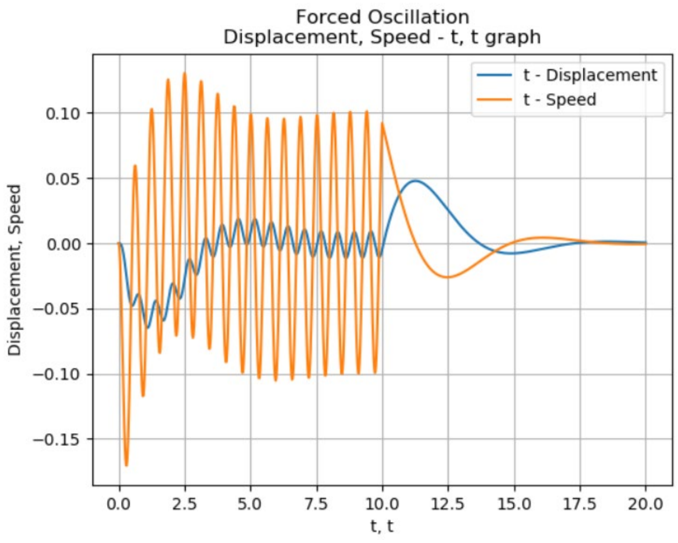
The **`.writePLOT`** method grpahs the result in the desired format.

`subj` governs which variables are graphed. It is a list of tuples containing variable names. The name that denotes time is **`'t'`**. For example, setting **`subj = [('t','x'),('t','y')]`** draws x-t, y-t graphs in the same graph. **`.writePLOT`** also supports 3d plotting. So **`subj`** can also contain tuples of length 3. Setting **`subj = [('t','x','y')]`**graphs the result on a 3D space. If nothing is given for **`subj`**, it graphs everything with respect to time in 2D.

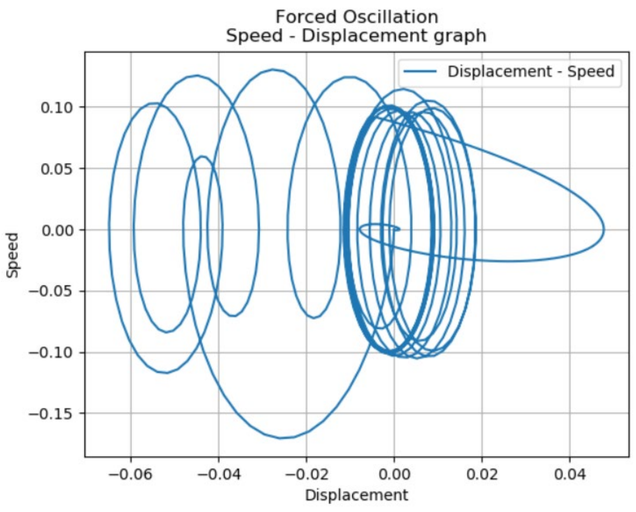
`points` govern how many points the whole graph is drawn. 1000 is recommended since the rendering could take extensive time. Setting it to 0 overrides the parameters and follows **`s`**, which is how many to skip (same as that of **`.writeCSV`**).

Giving more examples from the example mentioned in (2.),

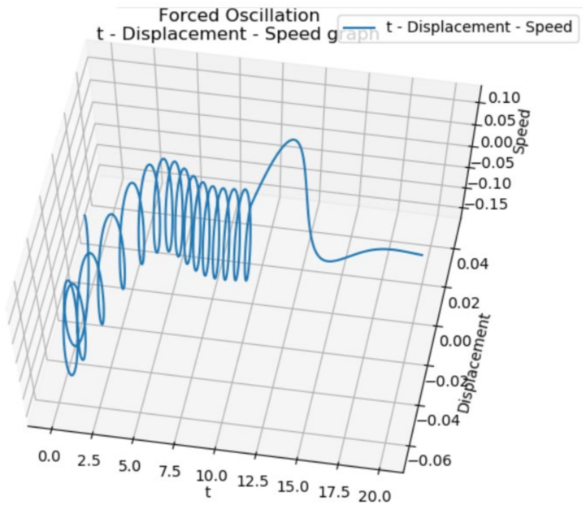
```
R.writePLOT()
```



```
R.writePLOT(subj = [('Displacement','Speed')]
```



```
R.writePLOT(subj = [('t','Displacement','Speed')]
```



.writeANIM(subj = None, timescale = 1, tail = 1, refreshRate = 20, quality = 100):

(from hashbrown 4.4.0+)

The **.writeANIM** method animates the result in a desired fashion. An important aspect of **.writeANIM** is that it animates entities only with respect to time (which is probably all that is needed for solutions of differential equations).

The **subj** parameter works the same with that of **.writePLOT**. It governs what is to be animated. **subj = [('x','y')]** will animate the variables with name **'x','y'** with respect to time. **.writePLOT** also supports multiple simultaneous animations and 3D animations (do try it!).

The **timescale** parameter determines how fast the animation is played with respect to real time. For instance, setting **timescale = 2** plays animation twice as fast as reality (assuming each the unit for time is 1 second)

The **tail** parameter rules the length of the tail behind the animation. If the animation scribbles too much, the **tail** parameter could be set smaller to clean up the animation. Setting **tail** to 0 Traces out the whole process without erasing any of it.

The **.writeANIM** method actually returns a `matplotlib.animation` object. So those familiar with `matplotlib` can further modify it in scope of `matplotlib` (save as video, etc.)

4. Summary

The functions and methods presented above are the critical parts of this program. They could each be encapsulated, giving extra versatility.

To summarize, the summary of the example is given.

```
from hashbrown import *
def interruptedForcedOsc(b,m,k,F0,w,tf,ts,te,name = 'ForceInt',step = 0.0001):
   def F(t):
      if t>=tf:
         return 0
      return F0

   def xp(t,x,y):
      return y
   def yp(t,x,y):
      return -(b/m)*y - (k/m)*x - (F(t)/m)*sin(w*t)

   F1 = Function(xp,3,'Displacement')
   F2 = Function(yp,3,'Speed')

   return rk4(ts,te,step,((0,0),(F1,F2)),name)

R = interruptedForcedOsc(1,1,1,10,1,10,0,20)

[additional [Result] actions (.get, .writePLOT, etc.)]
```

(the python file can also be found here: [Tutorial.py])

The `interruptedForcedOsc` function generates a solution that gives Displacement and Speed values when the Drag constant is `b`, mass is `m`, spring constant is `k`, driving force is a sinusoidal force of amplitude `F0` with the angular velocity `w`, that stops at `tf`. And it is simulated from `ts` to `te`. And this script uses the result to save a CSV file and draw a graph. The function itelf is in the tutorial. But representing the results is left for the user.

5. Additional Documentation

There are multiple methods and functions that were not covered in the documentation above. Although these features are not the major features, it is powerful in some mathematical and visual aspects.

- More on **`rk4(ts, te, step, s, name)`**

The full representation of the **`rk4`** function's parameters are

`rk4(ts, te, step, s, name, ti)`

and there are two additional features to this function.

1.Reverse calculation

If `ts` is greater than `te`, the function automatically assumes the initial value is actually given for the `te` and reverse calculates back to `ts`. So, the **`rk4`** function is actually capable of taking 'final conditions' to calculate the result

2.Midpoint calculation

The same is true with giving 'middle conditions.' This is what the parameter `ti` is for. The parameter denotes the time the initial condition applies. For example, setting `ts = 0, te = 10, ti = 5` finds a solution where the given conditions are true a t = 5. Since the process takes two calculations, two computations will take place. (one forward and one reverse)

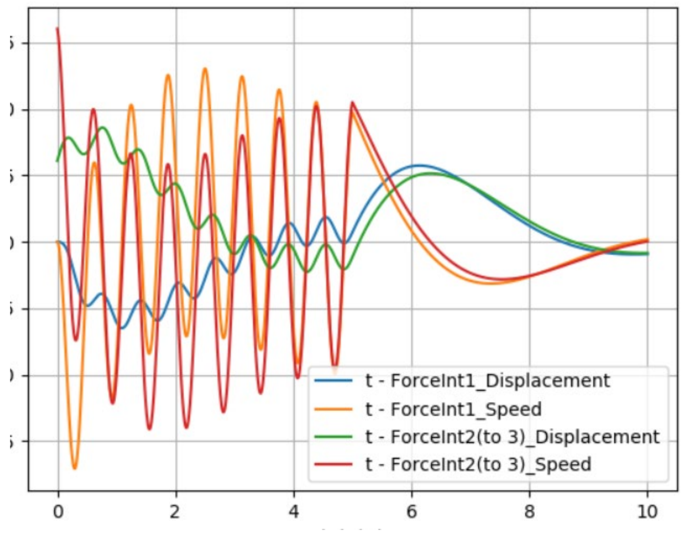
- **`mapF(ts,te,step,f,name)`**

mapF returns a **`[Result]`** class object of a simple time-based function. The parameter `f` takes a tuple of **`[Function]`** class elements to return the result as the value of that function form `ts` to `te` and by timestep `step`. Since it returns a **`[Result]`** class object, it could be represented as shown above in section 4.

- **`.merge(R)`**

The **`.merge(R)`** method returns a **`[Result]`** class object that merges two instances of results (`self, R`). If the time ranges are incompatible, the function simply pastes the first or last value as the result representing that time. Also if the timestep between two results are not compatible, It adopts the larger timestep as its own timestep. Although the merging for results are possible between for different timesteps, it is recommended that the one timestep is a multiple of another.

To avoid name collision, the merged values have their original names as prefixes. But any name with '_' will not be changed. Here is an example of a merged plot.



6. Legacy

Some functions and methods are no longer used, or a better alternative was introduced with new patches. The documentation for them are written here.

`- .animateF(subj = None, scale = 1, tail = 1):`

The **`.animateF`** method is an old, legacy method that animates the result. This method is kept for version consistency. It has slightly different parameters from **`.writeANIM`** :

The **`scale`** parameter determines how fast the animation is played. When **`.animateF`** was developed, it did not support ratio to realtime. Now, the scale parameter simply translates to **`timescale = 3*scale`**.

The stability and computation efficiency has significantly been improved from **`.animateF`** to **`.writeANIM`**. So, using **`.writeANIM`** is strongly recommended. In fact, **`.animateF`** is simply translated to:

```
        .writeANIM(subj,scale*3,tail)
```

References, Acknowledgement and Further Reading

All code in `hashbrown` was constructed independently.

The main mathematical principle behind this program, the Runge-Kutta 4th Order method, is referred from the lecture of Mathematical Modeling by Professor Yongdeuk Kim of the Korea Science Academy of KAIST. And all my thanks go to his teachings which gave birth to this library, and kept my interest in mathematics.

Thanks to Korea Science Academy 19-124 황제욱 and 19-057 여승현 for feedback and testing on version 4.3.0 and inspiring the update for 4.4.0

A concise explanation on implementation of Runge-Kutta 4th Order method is given by GeeksforGeeks.